CS422 - Programming Language Design

Concurrency and OO Homework

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

This homework assignment has two problems and is due on

Thursday, November 16, at midnight.

Problem 1: Concurrency

This problem requires you to modify the provided definition of FUN (see the file `funkConc.maude` in the provided zip archive) to change the semantics of some existing features in the presence of concurrency. If you prefer, you can do it for `KOOL` instead.

To do this problem, you have to handle

- the K definitions of the features that you defined or modified (no need for fancy typing or absolute rigor for the K definitions, you can write them by hand); and
- the new Maude definition.

The former will be needed to understand your solution, and the latter for checking it against some programs.

In the current definition of FUN or KOOL, no locks are automatically released in case of an exceptional change of the normal execution flow of the program (exceptions, return, break/continue): the programmer has the responsibility to release locks via appropriate coding. This might be annoying and cumbersome to achieve; advanced programming languages, e.g., Java, automatically release the resources acquired in a block when the block is exited, normally or abnormally.

Your task is to modify either FUN (the provided `funkConc.maude` file) or KOOL definitions to automatically release all the locks acquired during the execution of a function, a try or a loop block, when these are exited abruptly with a return, an exception throwing, or a break/continue, respectively.

Probably the best way to achieve this is by recording the locks held by a thread whenever saving the control stack, and restoring it in case of exceptional behavior. However, be warned that this

restoration should be approached with caution.

Example

```
let f = fun l ->
  (acquire(l) ;
  return(# 0))
in
  spawn(f(# 1);f(# 2)) ;
  spawn(f(# 2);f(# 1))
```

The above program should never reach a deadlock since locks ought to be released by return statements.

Problem 2: OO

Add Private Methods to KOOL

Currently, all methods in KOOL are **public**, meaning that any method can be invoked by any object of any class. We would like to extend KOOL with **private** methods. In our definition, a method marked `private` enforces the following restriction:

the target of the message send must be `self`

This is more restrictive in some ways and less restrictive in others than the **private** concept in other languages. Specifically,

- Methods marked `private` may be invoked from methods defined in subclasses; if class `C` defines `private` method `M`, method `N` in class `D`, where `D` is a subclass of `C`, can invoke `M`;

- methods marked `private` can only be invoked on the current object, meaning an object cannot invoke `private` methods on another object, even if that object is of the same class.

These are different than Java, for instance, where `private` methods are not visible in subclasses but *are* visible on other objects of the same class.

This is potentially a bit strange, but it goes well with our field lookup rules, which work in the same way – an object can see all of its own fields, even those it inherits. But, it cannot see fields in another object, even one of the same class. In some sense, this is a hybrid of `private` and `protected` from languages such as Java.

Implementation

To implement this, you will need to modify the semantics of KOOL. The syntax for `private` methods has already been added to both the SDF parser definition and the Maude syntax definition. It simply involves placing the reserved word `private` before the word `method` in the definition. Programs that encounter `private` methods will work, but will disregard the `private` attribute of the method.

To get private methods to work properly, you will need to:

- Extend the definition of a `METHOD` to include information on whether it is or is not private (look at module `METHOD` to see how other information is tracked)
- Capture this information when methods are processed (see specifically the module `METHOD-SEMANTICS`)

- Modify the send semantics in `SEND-SEMANTICS` to account for `private` methods. Specifically, a `private` method should only be invokable if the object ID of the current object is the same as the object ID of the object that initiated the message send (tracked in the `invoke` continuation item). If this is not the case, a `MethodPrivateException` should be thrown. Note that ordering of potential errors is important – if you have an invalid call on a private method that *also* has an invalid signature (i.e. the caller is trying to invoke a private method when it shouldn't be, and is doing so with the wrong number of arguments), you should throw a `MethodPrivateException`. The `InvalidSignatureException` should only be thrown when the method call is otherwise fine.

An Extension

Since it may seem odd to allow `private` methods to be invoked from subclasses, it seems tempting to change this. To do so, we can change our rules for `private` to match the following:

- Methods marked `private` can only be invoked from other methods defined in the same class
- methods marked `private` can only be invoked on the current object, meaning an object cannot invoke `private` methods on another object, even if that object is of the same class.

Here, the second point is the same, but the first has changed. This tightens `private` visibility, so subclasses can no longer see `private` methods in superclasses. Here, you will want to look specifically in `SEND-SEMANTICS` to change the rules. The most direct way is probably to modify the `invoke` continuation item to keep track of

the class that initiated the request, but there are other options as well.

To Start...

To get started, retrieve the implementation archive posted with this document. You will especially want to concentrate on the definition of KOOL in `c-kool.maude` (Concurrent KOOL). A webpage to parse programs is currently being built, to allow you to build your own test programs. For now, you can use the test programs in the `examples` directory. If you really want to parse programs on your own system, you will need to install several components of ASF+SDF: the ATerm library, the SDF parser, and Stratego. Contact Mark Hills for further information on this.